BAKKALAUREATSARBEIT

A parser for the Java programming language built with kdevelop-pg

ausgeführt am

Institut für Computersprachen,
an der Technischen Universität Wien

unter Anleitung von

A.o.Univ.Prof. DI. Dr. Franz Puntigam

durch

Jakob Petsovits

Erlengasse 13
A-7312 Horitschon

Wien, Januar 2006

Abstract

This document describes how to implement a parser for programming languages by using parser generators, shown by example of a new Java parser which was implemented for this thesis. The parser consists of a lexical analyzer created with Flex, a grammar for the kdevelop-pg parser generator. After providing an overview of parser concepts and kdevelop-pg functionality, the document depicts the process of creating the required components, covering challenges, strategies and common pitfalls of authoring a parser. It also describes shortcomings of the employed tools and short-term practical solutions as well as proposals to fix these problems cleanly in the long term.

Contents

1 Introduction

Parsers are a key element to processing structured data. Given textual input, they can perform checks on syntactic and semantic correctness and translate the original data into structures which can more easily be processed by the people or programs using the parser. There are different kinds of parsers for all kinds of data. Most of them handle either documents (such as XML data) or source code of programming languages (such as C++ or Java code) as input.

As opposed to natural ones, programming languages are designed in a way so that they can be parsed within reasonable boundaries of running time, universal validness and programming effort. Parsers processing programming languages are needed by all kinds of development tools, most importantly by compilers and development environments [1]. Compilers use them in order to translate source code into executables or machine-independent bytecode representation, while development environments visualize the gained data to assist the developer, for example with code completion or refactoring features.

1.1 Java support in KDevelop

KDevelop[1] is an integrated development environment for the K Desktop Environment (KDE)[2] available as open source[3] and has traditionally been focusing on C/C++ support. Nevertheless it also provides support for a range of other programming languages, where the feature sets and quality of the different language support parts vary greatly. Existing support for Java[4] includes a parser for version 1.2 of the language, a debugger, a class wizard and file templates, support for the Apache Ant build system[5] and problem reporting functionality. Code completion [13] is not available for Java, and despite of the broad feature set, KDevelop's Java support lacks refinement and a maintainer.

As of the time of writing, KDE is undergoing a major porting and refactoring effort. When it is done, KDE will base its libraries and applications on version 4 of the Qt Toolkit[6]. KDevelop participates in this effort, its whole codebase including the language support subsystem is being reworked and will result in KDevelop4 which is scheduled for late 2006. Due to the changes, language support parts have to be adapted too, which has not yet happened for the Java support.

The parser introduced in this document will provide a new basis for Java support in KDevelop, updating language recognition to Java 1.5 (also known as 5.0) and paving the way for advanced features like code completion and refactoring [9]. It is written with KDevelop integration in mind, therefore prefering performance, maintainability and error tolerance in case of doubt.

1.2 Concepts and capabilities of parsers

Common parsers first determine individual tokens by splitting up the input into seperate character sequences, which is known as lexical analysis. When this is done, they perform a syntactic check determining the grammatical structure and correctness of the tokens.

Most parser generators produce either LL(k) or LALR(1) parsers. LL(k) parsers belong to the family of top-down parsers, they feature small parsing tables and good debugging and error recovery facilities. LL(1) parsers are a subset of LL(k) ones, lacking the capability to look ahead for upcoming tokens. LALR(1) parsers, at the price of large parsing tables, allow more general applicability to programming languages, which makes them easier to write. They belong to the class of bottom-up parsers. Parser types and theory are discussed in detail in [2].

[1]KDevelop: http://www.kdevelop.org/
[2]K Desktop Environment (KDE): http://www.kde.org/
[3]Open Source Initiative: http://www.opensource.org/
[4]The Java programming language: http://java.sun.com/
[5]Apache Ant: http://ant.apache.org/
[6]Trolltech Qt: http://www.trolltech.com/products/qt/index.html

2 Parser generators

2.1 Parser generators vs. hand-written parsers

Parsers are ultimately just chunks of code for a given programming language. Compared to most other sources, they contain a great amount of similar code segments only differing in details. Together with the fact that lexical and grammatical structures can be defined more easily by using semantics (and optionally syntax) of the standardized EBNF [12], this resulted in the rise of parser generator software, also known as compiler-compilers.

In general, parser generator software takes a formal description of text structures and generates routines which are used to actually process the given text blocks. Available parser generators differ in the type of parsers that they generate, in the set of supported output languages, and features like error recovery facilities, inclusion of lexical analyzers, or syntax. There is a multitude of parser generators available[7], ANTLR[8] [14] and yacc/bison [15] being amonst the most popular ones.

2.2 The kdevelop-pg parser generator

2.2.1 Background and functionality

kdevelop-pg [9] was created by Roberto Raggi as a side product of his hand-written C++ parser for KDevelop4. It generates readable C++ code in the style of the C++ parser's sources and produces LL(1) type parsers that can be extended to LL(k) using manual look-ahead statements. It features automatic generation of abstract syntax trees (ASTs) and default visitor classes, following the Visitor pattern [10]. Lexical analysis is done by external code, preferably generated by Flex[10].

As a young project, it contained some bugs that had to be fixed before it generated correct code for the Java parser. Also, automatic error recovery and LL(k) capabilities are still missing and have to be implemented by extending the parser with custom extensions. Documentation for syntax and behaviour was not existing by now and is initially available in this document.

2.2.2 Syntax of grammar files

In order to generate the parser, kdevelop-pg needs an input file specifying the grammar rules of the parsed language. The rules are similar in expressivity to EBNF derivation rules, but the syntax in kdevelop-pg is different. First of all, all upper-case words are recognized as terminals (tokens), while other words represent non-terminal symbols (rule names).

The following table shows the rules that can be used within kdevelop-pg grammars:

Postfix sequences "`item @ sep_item`" are only an abbreviation for convenience and are translated to their original meaning "`item {sep_item item}`" by kdevelop-pg.

Annotations make up the structure of the AST. The conventional syntax (without "`#`") causes kdevelop-pg to create a member variable in the data structure of this rule, named identical to the variable name in the grammar.

Annotations prefixed with "`#`" are used to store item lists, for example lists of "statement" rules inside a statement block rule. In the generated code, kdevelop-pg creates a member variable named like the previously mentioned one, but with "`_sequence`" as suffix.

It is also possible to insert C++ source code into the grammar. C++ boolean expressions can be used to specify a condition for an item in an alternative items list so that it may only match if the condition is met. These conditions can be used as semantic or syntactic predicates. The most useful application of item conditions is the look-ahead macro which can retrieve upcoming tokens in advance. It then looks like this:

[7]List of parser generators in the Wikipedia article: http://en.wikipedia.org/wiki/Compiler-compiler
[8]The ANTLR parser generator: http://www.antlr.org/
[9]The kdevelop-pg parser generator: http://websvn.kde.org/branches/work/kdevelop-pg/?rev=497875
[10]Flex fast lexical analyser generator: http://www.gnu.org/software/flex/

Description	EBNF notation	kdevelop-pg notation
Rule production	rulename = item1 item2;	item1 item2 -> rulename;;
Epsilon (”nothing”)	not defined	0
Choice of alternative items	(item1 | item2)	(item1 | item2)
Optional items	[item1 item2]	(item1 item2 | 0)
Sequence of one or more items	(item)+	!(item)
Sequence of zero or more items	{item}	(!(item) | 0)
Postfix sequence	item {sep_item item}	item @ sep_item
Annotation, storing found items	not defined	variable=item
Annotation, storing item lists	not defined	!(#variable=item)
Line comment	(* commentary text *)	– commentary text

Table 1: kdevelop-pg grammar notation syntax

```
    (  ?[: LA(2).kind == Token_LPAREN :]
       method_name=identifier
       LPAREN method_arguments=argument_list RPAREN
     |
       variable_name=qualified_identifier
    )
```

Code that is not used as a condition can be inserted after each item (what is known as semantic actions):

```
       GREATER_THAN     [: ltCounter -= 1; :]   -- ">"
     | SIGNED_RSHIFT    [: ltCounter -= 2; :]   -- ">>"
     | UNSIGNED_RSHIFT [: ltCounter -= 3; :]   -- ">>>"
    -> type_arguments_or_parameters_end ;;
```

Source code segments can span multiple lines. It is also possible to place a source code segment at the beginning of the grammar file, in that case it is inserted into the generated .h file. This can be used to define global variables, classes and other constructs.

2.2.3 Warnings and parsing behaviour

Being an LL(1) parser generator, kdevelop-pg creates parsers that determine their way through the token stream based only on the currently processed token. There are no magic non-determinism solving algorithms, so it is up to the grammar author to keep the grammar as deterministic as possible.

kdevelop-pg recognizes when there is more than one possible way to proceed for a given token in a specific parser state and subsequently puts out a conflict warning. There are two types of conflict warnings:

FIRST/FIRST conflicts indicate that it is not clear which item to select if a choice from two or more alternative items has to be made. Many of these conflicts can be solved by left-factoring or look-ahead techniques. Look-ahead is not considered by the conflict recognizer so even if the non-determinism is resolved, the conflict is still reported. This type of conflicts is handled by kdevelop-pg by selecting the first possible choice.

FIRST/FOLLOW conflicts incidate that it is not clear if the current (optional) item or the following one should be chosen. A common example for this type of conflict is the “dangling else” where it is unclear if an occurring “else” belongs to the currently processed “if” statement rule or to a higher-level one. kdevelop-pg handles this type of conflicts by acting greedy, which means that out of multiple possible choices, the leftmost (inner) one is selected. This is the correct behaviour in the great majority of cases.

2.2.4 Generated code

The code generated by kdevelop-pg can be divided into three sections:

The AST consists of node structures containing child nodes for every annotated non-terminal symbol in the grammar, and `std::size_t` integers for every annotated token, where the integer value holds the index of the token in the token stream. When traversing the AST, the token type can be retrieved with the expression

```
    token_stream->token(node->token).kind
```

which returns one of the `Token_*` values in the parser class's `token_type_enum` enumeration. All node structures are defined in the generated .h file.

The parser class expects a token stream that is already filled with recognized tokens. Filling the token stream has to be done by calling the lexical analyzer before starting the parser. The implementation of the parser class is contained in the generated .cpp file and has a generated method for every rule in the grammar. Annotated symbols and tokens cause the respective value in the AST to be filled if they are encountered. Node pointers and token index values that are not encountered when traversing the rule are initialized with 0 (zero) as default value. In summary, these methods make up for a recursive descent parser that processes one token after another.

If none of the currently available rules match, the parser class marks an error by calling (optionally) the `yy_expected_token()` and (in any case) the `yy_expected_symbol()` methods that have to be defined by the developer using the parser. These methods can be used to implement custom error recovery. (The parser recovers if the method's return value is true.)

The parser class can be used by calling any of the `parse_*()` methods, it is common to call the topmost rule which can derive the whole text document. The `parse_*()` methods return true if a valid token sequence is found, if the topmost rule returns true then the text input is fully matching the grammar.

In the header file, the declaration of the parser class also defines enumeration values for tokens (`token_type_enum`) and various helper methods such as the inline `yylex()` method which in fact has nothing to do with the one generated by Flex and consumes the next token in the token stream.

The default visitor class is meant to be subclassed and provides facilities to traverse the AST structure when parsing is done. Subclasses can extend the default visitor's traversing functionality by printing out the AST structure, performing semantic checks or storing data from the AST in the code model, like class and variable declarations.

2.2.5 Internals

The kdevelop-pg sources are very similar in style to the generated code as kdevelop-pg essentially implements a parser too, taking grammar files as input. The Visitor pattern is used extensively for computing closures, checking the input for conflicts and generating code output.

The parser for the grammar file is implemented with yacc functionality and fills the AST representing the grammar. When done, it computes the FIRST set which contains the possible start tokens of every item, that is terminals, symbols and combined ones such as alternative items or concatenation items). Also the FOLLOW set is computed for every rule in the grammar, it contains all possible tokens that can follow on the respective rule.

These two sets are the basis for everything else in kdevelop-pg. They are used to check on conflicts between rules, to tell if an item can derive to the "0" rule (called epsilon) and to determine the conditions that are put into the if and while statements of the generated parser class. All of these actions happen inside a visit_*() method of default_visitor subclasses.

After generating the code, kdevelop-pg uses an internal copy of the Artistic Style source code formatter[11] for optimizing readability of the output files.

[11] Artistic Style (also known as AStyle): http://sourceforge.net/projects/astyle/

3 Related works

3.1 Available Java parsers

Java is a popular language and is implemented by a good number of parsers. Depending on customs and practice of the software that they are written for, Java parsers come in a wide range of different flavors. The following ones have been found while researching the topic:

A grammar for ANTLR (subsequently referenced as "the ANTLR grammar") written by John Mitchell, Terence Parr and others. This grammar was put into the public domain and is available in different versions. The version used for reference when creating the new parser was published by Michael Studman[12] and creates an LL(k) parser for Java 1.5.

A grammar for JavaCC recognizing Java 1.5 with LL(k) traits, written and copyrighted by Sun[13]. It is written in a clean style and resembles the proposed grammar from the Java Language Specification [3] very much. I would have transcribed this grammar instead of the ANTLR one, but due to the unclear licensing this was conflicting with KDevelop's open source licenses and therefore not feasible.

A grammar for JikesPG used by Eclipse[14] for generating its own Java 1.5 parser. It generates an LALR(1) parser which is consecutively used inside the internal Java compiler that Eclipse runs as background task every time a file has changed. The parser is available under the Eclipse Public license, its sources are a bit exhausting to read. Another JikesPG grammar for Java 1.5 is used by the Jikes compiler itself[15] and subject to the terms of the IBM Public License[16].

A grammar for YACC used by the gcj Java compiler[17] from the GNU Compiler Collection. This grammar stays close to the proposed grammar in the Java Language Specification, recognizes Java 1.4, and is licensed under the GNU GPL[18] version 2 or later. As a YACC grammar, it generates an LALR(1) parser.

[12]An ANTLR grammar for Java 1.5: http://www.antlr.org/grammar/1090713067533/index.html

[13]Sun's Java grammar for JavaCC: https://javacc.dev.java.net/servlets/ProjectDocumentView?documentID=3131&showInfo=true

[14]Eclipse's Java grammar for JikesPG: http://dev.eclipse.org/viewcvs/index.cgi/org.eclipse.jdt.core/grammar/java_1_5.g?rev=1.59&content-type=text/vnd.viewcvs-markup

[15]Jikes' Java grammar for JikesPG: http://cvs.sourceforge.net/viewcvs.py/jikes/jikes/src/java.g?rev=1.48&view=auto

[16]IBM Public License: http://www-128.ibm.com/developerworks/library/os-ipl.html

[17]gcj's Java grammar for YACC: http://gcc.gnu.org/viewcvs/trunk/gcc/java/parse.y?view=markup&rev=108608

[18]The GNU General Public License (GPL): http://www.gnu.org/copyleft/gpl.html

4 How it works

4.1 Prerequisites

4.1.1 Traits of the Java programming language

The object-oriented programming language Java was developed by Sun Microsystems[19] and is available to the public since 1995. It is possible to write LR(1) grammars for Java, but there are a few problems for LL(1) grammars that are believed to be impossible to solve without workarounds. This makes it necessary to implement solutions for those problems using different approaches than pure code generation from the grammar.

Also, the requirements for Java parsers have changed with the different versions of Java. From a parser point of view, Java 1.4 lacks support for generics, enhanced for statements (for-each), native enumeration types, variable-length argument lists and annotations in comparison to 1.5. Ignoring API changes, Java 1.4 only adds support for the "assert" statement to the features of version 1.3.

4.1.2 Using existing parsers as foundation

Creating a grammar from scratch is a remarkably big effort which is hardly reasonable to do except if there is no access to existing reusable grammars. Today, there are grammars and lexical analyzers for many languages freely available on the Internet, so it is advisable to evaluate and select from existing ones.

The Java parser herein is based mainly on two sources: the reference grammar in the Java Language Specification (JLS) [3] and the ANTLR grammar mentioned in section 3.1. The JLS also provides textual description and another grammar distributed among all its chapters to clearly specify the exact meanings of every language element. Comparing and understanding the intentions of the same grammar element in both references does help a lot when encountering cases where one of them poses doubts about meaning and correctness.

As the created parser is intended for use inside KDevelop, licensing is also an issue. There was no verbatim copying but it is not sure how exact transcriptions of grammar elements are viewed by the law, so the best idea is certainly to avoid possible copyright infringements in the first place. The JLS grammar explicitely states that it can be used by tool authors, while the ANTLR grammar implies this with declaring itself as being in the public domain.

4.2 Creating the framework

The parser generator doesn't create code for command line option handling, text input and output, it just operates on a token stream that has to be provided from outside. The framework takes care of this, tokenizing the input with calls to the lexical analyzer (using `yylex()`) and invoking the parser afterwards. It also incorporates enhancements of the error output by printing a number of tokens before and after the token which triggers a parse error.

The importance of the framework is limited, because these mechanisms will be adapted to the container application using the parser later on. The framework used here is therefore merely a test container, good enough to ensure correctness of the parser, but with an eye on being obsoleted through integration into KDevelop.

4.3 Creating the lexical analyzer

Parsers generated by kdevelop-pg operate independent from the lexical analyzer. The only condition that has to be met is a token stream containing values from the `token_type_enum` enumeration defined in the generated parser class. For this reason, the lexical analyzer must be able to return custom defined token values for lexical matches.

Flex[20] satisfies this requirement and is widely known amongst programmers, which makes it a good choice for a parser that is intended for use within open source software. Flex is

[19]Sun Microsystems: http://www.sun.com
[20]Flex fast lexical analyser generator: http://www.gnu.org/software/flex/

the only popular lexical analyzer that generates C code so it can easily be used within the C++ framework.

Not being able to find an existing Flex file for lexing Java tokens, it had to be written from scratch. Such a file that was contained in ANTLR's collection of examples was found when approaching the completeness of the parser, it was merged into the new one for better looks and maintainability while not enhancing the then-existing functionality.

4.3.1 Lexical Analysis with Flex

The JLS contains a seperate chapter covering lexical analysis so transcribing the specification to the Flex file format posed no serious difficulties. Though it is possible to split up more complex lexemes into smaller parts and use Flex states to process the parts individually, this approach has the disadvantage that the corresponding input text is also split. In most cases it will be better to construct one single matching pattern instead to avoid the need for a self-managed string buffer. Nevertheless, comments do not need being stored as token so block comments only gain from the clarity of state driven matching.

Despite the general ease of use, one problem worth mentioning was discovered in Flex concerning its handling of newline characters. With its "`.`" character, Flex offers an abbreviation for the "`[^\n]`" pattern which matches all bytes except the POSIX newline character. This usually doesn't cause problems on Unix/Linux systems where `'\n'` is the common newline delimiter. On the other hand, there are different kinds of delimiters used by Windows systems "(`\r\n`)" and by Macintosh systems (`'\r'`). When using "`.`", Flex doesn't take into account that `'\r'` can also be a newline character, which causes incorrectly matched character sequences and parsing errors in less common cases.

Considering this, it is advisable to replace all occurrences of "`.`" with "`[^\r\n]`" when striving for total correctness also for text files containing non-Unix newline characters. For example, the correct pattern for line comments looks like this:

```
"//"[^\r\n]*    /* line comments, skip */ ;
```

Another question is what to do with invalid tokens. One possibility would be the use of `ASSERT(0);` statements when encountering such tokens, but doing so prevents following error messages from being shown and makes parser calls impossible, even if it had appropriate error recovery capabilities.

The solution employed by the lexical analyzer is to introduce another token called `Token_INVALID`. It is returned when no other pattern can be matched, it's easy to recognize for the parser and it doesn't conflict with existing tokens. Before returning the `INVALID` token, it is possible to put out a warning to the problem reporting function.

Invalid strings and character escapes can be determined with a pattern allowing all escape sequences instead of just the specified ones and recognizing newline characters in unclosed strings. With this information it is possible to produce better warnings and at the same time keep the parser unconfused. When employing this technique, it is important to make sure that the pattern for invalid tokens doesn't match valid strings and at the same time exceed their lengths - this would cause false positives. As an example, the string patterns in the Java lexer are shown here:

```
[\"]({Escape}|{Multibyte}|[^\\\r\n\"])*[\"]  return java::Token_STRING_LITERAL;
[\"]({Escape}|{Multibyte}|[\\][^\\\r\n\"]|[^\\\r\n\"])*(([\\]?([\r]|[\n]))|[\"])
{
    reportProblem("Invalid string literal...");
    std::cerr << yytext << std::endl;
    return java::Token_STRING_LITERAL;
}
```

4.3.2 Unicode support

The JLS specifies support for Unicode [11] all throughout input files. Not only is it possible to use any Unicode characters (native or though escape sequences) inside string and character literals but also in the code itself. The keywords, as strictly defined words, can of course not contain Unicode characters, but identifiers can. This makes it possible for programmers to use characters of their own language (for example “è”, “ä” or even Chinese characters) inside variable, method and class names. Characters are valid identifier parts if the corresponding Java methods return true for them.

Naturally this poses a problem for the Flex/C(++) approach, because C and C++ do not support Unicode without capable string libraries, and even if they did it would not be feasible to call a Java method from there. As a result, it was necessary to retrieve the Unicode values for the specified letter and digit classes and afterwards constructing Flex patterns for these characters.

Converting Unicode values to Flex patterns is made possible with a Haskell script by Hans Aberg [21] written specifically for this case. Assisted by the script, the Flex file now contains 19 patterns that look like the following example and cover all Unicode letters, digits and other multi-byte characters:

```
 /* \u00c0-\u00d6, \u00d8-\u00f6, \u00f8-\u00ff */
Letter2         [\xC3]([\x80-\x96]|[\x98-\xB6]|[\xB8-\xBF])
```

The different patterns can be combined to make the definition of Java identifiers look as simple as this:

```
{Letter}({Letter}|{Digit})*  return java::Token_IDENTIFIER;
```

This approach allows for matching files that are encoded with UTF-8. Other encodings cannot be matched and must be transformed to UTF-8 before passing it to the lexical analyzer. When the parser is integrated with KDevelop, this won’t be much of a problem because the underlying Qt Toolkit’s QString class uses UTF-8 as its native encoding for all strings.

The current implementation of Unicode support has another drawback: Unicode escape sequences (for example, `\u0020` representing a whitespace) are only accepted inside string or character literals, but not as keywords or as identifier parts. The JLS specifies that all unicode escapes are to be transformed into their character counterparts, which would require call a preprocessor on the input file before starting the actual lexical analysis. For performance reasons and because of the very rare usage of Unicode escapes outside strings and character literals, it has been decided not to implement this feature for the Java parser.

4.3.3 Testing the lexical analyzer

As a prerequisite for testing the lexer, we must make sure to define the token values that are used there. The required header file is generated by kdevelop-pg, so we need a preliminary grammar file that contains all the token names in any of its rules. When testing the lexer, it makes sense just to have a start rule which contains all tokens in an alternative item.

Testing mainly consists of a collection of test cases and manually verifying that the recognized tokens have been matched correctly. Debug output is easy to create as it just requires writing the `yytext` variable to the output stream each time `yylex()` has been called. When the tokens are seperated by a whitespace or newline character, it is easy to see which ones have been matched incorrectly.

The tests for the lexical analyzer can be done independently from testing the rest of the parser, which is described in section 4.7 further below.

4.4 Transcribing the grammar

4.4.1 Strategies

Merging two grammars requires to consider which one does it better and then include the best of both worlds in the new one. Having two references was a good idea as they complement each other well. The JLS grammar provides clear rule names and is easy to understand

[21] Unicode Flex Character Classes: Conversion script to create regular expressions for Flex by Hans Aberg, http://lists.gnu.org/archive/html/help-flex/2005-01/msg00043.html

in syntax and meaning, but doesn't address solutions for LL-based parsing. The ANTLR grammar is written for practical application and therefore provides good ideas for conflict solutions and refactoring, but its concepts are a bit harder to grasp at times. Despite these differences, both grammars resemble each other quite a lot in overall structure.

It proved to be a good idea to stay close to JLS like rule naming and choose the better one of two approaches on a case by case basis. ANTLR's action blocks (the ones involving a lot of "#" signs) were simply dismissed in most cases because kdevelop-pg claims their task of filling the AST structures for itself. Making usage of kdevelop-pg's postfix operator can also simplify the grammar and flatten the generated AST – it is possible to write `#expression=expression @ COMMA` inside kdevelop-pg rules instead of the following ANTLR rule (which is a bit lengthier) without losing information:

```
// This is a list of expressions.
expressionList
    :     expression (COMMA! expression)*
          {#expressionList = #(#[ELIST,"ELIST"], expressionList);}
    ;
```

In order to keep the generated AST flat and uncluttered, it is generally a good idea to reuse rules as much as possible, not to implement them twice and to prefer sequences over nesting. kdevelop-pg generates an AST node for every rule in the grammar, so heavy nesting results in a more complex AST structure. However it was sometimes necessary to create nearly similar rules in order to keep the conflict count low, to ensure correctness or to prevent performance losses.

When rules are required by other ones but not yet implemented, it will prove worthwhile to track the missing rules with a list of placeholders:

```
STUB_A -> conditional_expression ;;
STUB_B -> statement ;;
STUB_C -> type_parameters ;;
(...)
```

In any case it is a good idea to document identified conflicts and to comment unclear sections, so that the grammar easier to understand by future contributors or when remembering the original intentions is hard.

4.4.2 Conflict warnings

The Java grammar consists of two big parts that can be seperated quite well from each other. One part is the high-level structure covering rules for type and method definition, and the other one are more low-level rules that cover the "statement" rule and its children. Both parts require to call some of the other part's rules, but these can easily be provided by placeholder rules. Splitting the grammar into pieces keeps the number of rules low, which in consequence causes less errors for the seperate parts. If new conflict warnings arise, this can be an advantage as it's easier to find them.

The remaining conflicts are mostly quite straightforward to analyze and fix. First/follow conflicts that originate from optional items are harmless throughout the whole Java grammar, which probably also applies to most other languages. They are only dangerous when loops are involved, which is every occurrence of the "`!(...)`" or the postfix operator in kdevelop-pg.

First/first conflicts come in bigger numbers and always need to be resolved manually by employing techniques covered in the two upcoming sections 4.4.3 and 4.4.4.

4.4.3 Refactoring

In LL(1) parsers, rules and sub-rules are chosen depending on their first token. If there are two rules or sub-rules that start with the same token, there is a first/first conflict between those two. The ANTLR parser generator provides advanced functionality to automatically resolve such conflicts where possible, so that this is a conflict-free ANTLR rule:

```
postfixExpressionExcerpt
    :    DOT^ "this"
    |    DOT^ newExpression
    ;
```

kdevelop-pg lacks these resolution mechanisms. If conflicts ought to be avoided, the corresponding rule in kdevelop-pg must look like this:

```
DOT (THIS | newExpression) -> postfixExpressionExcerpt ;;
```

Resolving such conflicts is known as left-factoring. It's not always as easy as in this example because often the identical start tokens are distributed among different rules. These cases are more difficult to overcome as also the meaning of rules may change, which can make it necessary to also adapt other rules additionally to the intended ones.

Even if the ANTLR grammar is well thought out in most places, not all of its constructs are optimal. For conflict prevention, AST structure improvement and correctness reasons, the handling of the rules for the primary expression and the "for" statement have been refactored completely. In general, refactoring of rules is recommendable if the current approach doesn't work or scale anymore.

4.4.4 Look-ahead

To overcome the shortcomings of pure LL(1) parsers, the kdevelop-pg generated parser class contains a method called `LA(int)` which retrieves upcoming tokens without consuming them. `LA(1)` would return the current token, `LA(2)` the next one, effectively it is possible to retrieve any token up to the last one called EOF. It's usually not necessary to check on `LA(1).kind` (which is equivalent to `yytoken`) because the conditions generated by kdevelop-pg already handle this by themselves. `LA(2)` on the other hand is very useful and helps resolving most conflicts that can't be solved with LL(1). Look-ahead is commonly used as shown in this example:

```
  ( ( ?[: LA(2).kind == Token_ASSIGN :]
      #value_pair=annotation_element_value_pair @ COMMA
    )
  | element_value=annotation_element_value
  )
-> annotation_arguments ;;
```

The `.kind` accessor and the `Token_` prefix are necessary because it is C++ code which is directly inserted into the generated condition.

Conditions with fixed `LA(3)` look-ahead or more are not needed at all for the Java parser. Nevertheless there are cases that can't be solved with any fixed look-ahead, these are described in section 4.5.2.

4.5 Extending the grammar with hand-written code

Especially as kdevelop-pg is missing certain functionality, it is especially important to be able to implement wanted features manually. This is made possible with C++ code that can be placed virtually anywhere. While it would even be possible to integrate semantic checks into the parsing process, it might be a good idea to strenthen modularity and check on semantics in a seperate class derived from the default visitor. The following uses of C++ code only support the parsing process itself.

4.5.1 Custom conditions

C++ conditions are not restricted to look-ahead only, they are also capable of checks on traditional variables. The Java grammar poses a special challenge with its specification of type arguments, which comes into play when multiple nested type arguments are used.

Because of the weak specification of closing greater-than characters there is an ambiguity between type arguments' closing signs and right-shift operators.

This ambiguity can be solved by allowing either one of "`>`", "`>>`" and "`>>>`" as closing signs. In order to work, this approach is combined with a counter variable tracking the number of opened type arguments, it is called `ltCounter`. For better insight, here are two of the rules where counting occurs, one of them uses the variables in custom C++ conditions:

```
   LESS_THAN [: int currentLtLevel = ltCounter; ltCounter++; :]
   #type_parameter=type_parameter @ COMMA
   (
      type_arguments_or_parameters_end
      -- make sure we have gobbled up enough '>' characters
      -- if we are at the "top level" of nested type_parameters productions
      [: if( currentLtLevel == 0 && ltCounter != currentLtLevel )
          { return false; }
       :]
    |
      0
   )
-> type_parameters ;;

   GREATER_THAN     [: ltCounter -= 1; :]  -- ">"
 | SIGNED_RSHIFT    [: ltCounter -= 2; :]  -- ">>"
 | UNSIGNED_RSHIFT  [: ltCounter -= 3; :]  -- ">>>"
-> type_arguments_or_parameters_end ;;
```

The "**`return false;`**" statement is not the best idea for use inside a grammar as it involves thinking at the code level, but kdevelop-pg currently doesn't provide another possibility for rules to fail manually. Using hand-written code, this is at least made possible at all.

4.5.2 LL(k) look-ahead

In the Java grammar, there are four rules that need LL(k) look-ahead with unknown, variable k. These cases are more difficult than the LL(2) ones as they require tracking whole rules in order to determine which path is coming up next. Because of the complexity involved, simple `LA(x)` conditions don't suffice here. The problematic conflicts that we're speaking of are these:

- The conflict between the `package_description` and `type_description` rules, both can start with any number of annotations that can be arbitrarily long. Only after encountering either the "`package`" token that belongs to `package_description` or one of "`class`", "`interface`" or others that belong to `type_description`, it can be decided which rule to descend into.
- The conflict between `expression` and `variable_declaration` exists twice in the grammar. Both rules can start with arbitrarily long type specifications like "`java.lang.TreeMap<int,string>`". Refactoring is virtually impossible because even if it was doable, it would seriously complicate the generated AST in most sensitive places.
- The conflict between type casts and expressions in parentheses is quite similar to the above as it also involves checking on type specifications. Some of the tokens following the ones inside the parentheses can also be similar in both rules.

All of these problems can be resolved by creating a `java_lookahead` class which is an adapted copy of the generated parser class. The look-ahead class uses the generated code to also parse the upcoming tokens, the difference is that it doesn't populate the AST structure and doesn't consume the tokens in question.

In the look-ahead class, every usage of AST node structures has been removed and the `yylex()` method is modified to use the `LA()` macro instead of increasing the token counter. It incorporates a good amount of `parse_*()` methods from the parser class, but leaves out the ones that are not required for the problematic cases to work.

The `expression` rule and nearly all of its children are amongst the `parse_*()` methods that were skipped for the lookahead class. Incorporating these methods would result in a lot more adapting and maintaining effort, but some of the problematic cases refer to the `expression` rule. As a workaround, occurrences of the `expression` rule have been replaced with simple counting of parentheses or type argument brackets, everything inside those tokens is simply skipped. (Luckily, expressions are not needed outside parentheses or type arguments for the cases in question.)

In the long run it would certainly be more productive if look-ahead code with unbounded LL(k) capabilities was generated by kdevelop-pg itself, as maintaining such look-ahead classes is error-prone and only advisable if the grammar is no more subject to change.

4.5.3 Supporting different Java versions

The new parser was written for compliance with the Java 1.5 standard. Nevertheless, with a few changes it is also possible to adapt the parser so that it accepts previous Java versions instead. This is done by introducing a new variable set by the framework that can be checked on by the lexer and the parser class. It is defined as an enumeration type in the grammar's section for code that is written to the C++ header:

```
enum java_compatibility_mode {
  java13_compatibility = 130,
  java14_compatibility = 140,
  java15_compatibility = 150,
};

class java_settings {
  public:
    static java_compatibility_mode _M_compatibility_mode;
};

static java_compatibility_mode compatibility_mode() {
  return java_settings::_M_compatibility_mode;
}
static void set_compatibility_mode( java_compatibility_mode mode ) {
  java_settings::_M_compatibility_mode = mode;
}
```

The initialization of this variable has to be done in an implementation file because of technical limitations of the C++ language. The Java parser currently initializes the compatibility mode variable in the .cpp file of the look-ahead helper class, but kdevelop-pg support for parser class members (as described in section 5.1.1) is the only proper solution in the long term and is greatly anticipated.

Depending on the value of this variable, it can be decided which tokens or token sequences are accepted. This is an excerpt from the lexical analyzer:

```
"enum"          {
    if (compatibility_mode() >= java15_compatibility)
      return java::Token_ENUM;
    else
      return java::Token_IDENTIFIER;
}
```

In the grammar, C++ conditions can be used to influence the parsing process:

```
   identifier=identifier
   (  ?[: compatibility_mode() >= java15_compatibility :]
      type_arguments=type_arguments
    | 0
   )
-> class_or_interface_type_part ;;
```

4.6 Issues with kdevelop-pg

Additionally to the problems mentioned in section 4.5.2, working with kdevelop-pg showed a few other shortcomings in the parser generator's functionality as well.

4.6.1 Exiting loops

In most cases, the postfix and "`!(...)`" operators are doing the right thing as they stop matching when a token other than the expected one is found. However, there are constructs in the Java grammar where a possibility to manually exit these loops would be helpful.

kdevelop-pg does not provide such a possibility with its regular grammar syntax, so one has to look at the implementation and resolve these issues using C++ code segments:

```
        #name=identifier
        (  !( DOT ( #name=identifier | star=STAR [: break; :] ) )
         | 0
        )
    -> qualified_identifier_with_optional_star ;;
```

This works because the postfix and the "`!(...)`" operator are transformed to a `while` loop when the parser code is generated.

If a condition has to be checked at the beginning of every loop, there is a way to emulate real conditions (which are not capable of ending loops) with code segments belonging to an empty epsilon rule:

```
        #name=identifier
        ( !( 0 [: if (LA(2).kind != Token_IDENTIFIER) { break; } :]
             DOT #name=identifier )
          | 0
        )
    -> qualified_identifier_safe ;;
```

Making use of these constructs should be avoided if possible because they require deeper knowledge of how the rules are transformed to code. If there are changes in the way the code is generated then these constructs can easily break and cause compilation errors, or even worse, unexpected behaviour.

4.6.2 Code generation bugs

In the parser class, conditions generated by kdevelop-pg check for each token that can be expected to occur in the current parser state, conforming to the corresponding rule in the grammar. These conditions are generated from the FIRST and FOLLOW sets which are computed prior to code generation [2].

kdevelop-pg contained a bug that resulted in erroneous parser code and was caused by incorrect computation of the FIRST set when the epsilon rule was involved. For example, kdevelop-pg correctly computes the FIRST sets of the two seperate rule elements "`(A | 0)`" (where the FIRST set is `[A,0]`) and "`B`" (where the FIRST set is `[B]`). When they were combined to "`(A | 0) B`", kdevelop-pg simply merged the two original sets, resulting in the combined FIRST set `[A,B,0]`. This is wrong because epsilon (the "`0`" rule) is only kept within the merged set if both of the original ones can derive to epsilon. The correct FIRST set for this example rule would therefore be `[A,B]` instead.

This problem was solved by extending kdevelop-pg with a mechanism to prevent merging epsilon into the FIRST set in these cases. kdevelop-pg's calculation of the set was changed so that it recursively computes the right-hand side of the currently processed rule element (that would be "`B`" and all possible rule elements right to it). When the right-hand side's FIRST set is computed, there is a check if it can derive to epsilon, and if it can't then merging a possible epsilon from the left-hand side is blocked. This fix not only enables kdevelop-pg to generate correct code but also dismisses most of the pointless conflict warnings that were caused by the incorrect FIRST sets.

For total correctness, two additional patches were needed. One of them fixes a problem where the wrong node in kdevelop-pg's parsing tree was used to generate test conditions, and the other one ensures that alternation items containing conditions make the rule fail when none of the conditions is met.

4.7 Testing the parser

Even when written by experienced authors it's very probable that newly created grammars as large as the Java grammar contain a number of bugs. As a consequence, testing the generated parser is just as essential as the actual creation process itself.

4.7.1 Creating a code reconstruction utility

When the grammar is complete, we want to know if the information that is gathered about the input text contains all that is required for further processing the data structures. This can be achieved with a test program that uses the AST which contains all the information retrieved by the parser. Traversing the AST, the test program reconstructs the original Java source code and subsequently proves both the correctness and the usefulness of this data.

The program, named javawalk, benefits from the already available default visitor class generated by kdevelop-pg. For common cases, this class can be derived and extended with code that further processes data from the AST. In case of the code reconstruction utility this is not possible because we need to process the concerned nodes in order and insert additional output between some of the node visitor calls. The javawalk utility modifies the `java_default_visitor` class by inserting appropriate print statements. The resulting class is called `java_print_visitor`, which is the heart of the program and only derives from the visitor interface called `java_visitor`.

The `java_print_visitor` class incorporates simple formatting so that the output code is not only semantically identical to the input but also readable. Some elements of the input file, like comments, empty statements or other unnecessary tokens that were not captured by the parser have to be dismissed though. If the grammar is changed then required modifications can still be tracked by finding out the differences between the newly generated `java_default_visitor` and the old one, the changes can then be transferred to the `java_print_visitor` class. Even though it's doable to keep the print visitor intact, it is advisable not to write such a class until the grammar is relatively mature, because these small changes are causing a high maintenance effort.

4.7.2 Gathering test cases

We want to prove that the parser works on most (if not all) correct Java files. Fortunately, there are many Java projects freely available on the Internet, so apart from a few self-made test cases the majority of files are taken from real world open source projects. Different types of projects have been selected as test cases, fetched from their respective source repositories:

- Eclipse[22] consists of a huge amount of Java code. The chosen test cases are just the files from the "platform" module of the Eclipse CVS repository [23], disregarding other modules like the "tools", "webtools" or "technology" projects. Nevertheless, with a total of 35287 files this is by far the biggest part of the test file collection. This number also includes over 600 invalid source files that had to be removed in order to get accurate testing results. (Those files are indeed test cases for Eclipse's parser, refactoring framework and other components, and fail parser runs by design.) Two folders with big amounts of invalid source files have been deleted together with other correct ones. The sources obtained from Eclipse contain files for different Java versions from 1.3 up to 1.5, so it offers a great testing ground for the parser.
- Other production-quality projects, though not as voluminous as Eclipse, have also been added as test cases. These are the complete sources of Hibernate3[24], the gcj compiler's reimplentation of the Java runtime library[25] and the ANTLR parser generator.
- Furthermore, the test cases have to include lower-profile projects as well, revealing challenges which result from poorer coding standards. This category consists of the mid-size zdt Chinese learning application[26], the mAkStats Hattrick statistics tool[27] and the HICS Hotel management program[28]. These projects have been selected randomly, two of them showed that there are indeed sources that make use of non-Latin-1

[22]Eclipse: http://www.eclipse.org/
[23]Getting the "platform" module from Eclipse CVS: http://wiki.eclipse.org/index.php/CVS_Howto
[24]Hibernate: http://www.hibernate.org/
[25]The gcj Java compiler and class library: http://gcc.gnu.org/java/
[26]zdt Zhongwen Development Tool: http://zdt.sourceforge.net/
[27]mAkStats: http://sourceforge.net/projects/makstats
[28]HICS Hotel Information and Control Services: http://developer.berlios.de/projects/hics/

Unicode characters (which were an issue for the lexical analyzer, as discussed in section 4.3.2).

Once the files have been gathered, they can be processed by the parser. However it's not feasible to manually run the parser on 40,000 files.

4.7.3 Batch processing

In order to handle the big amount of test cases, an automated test mechanism is needed. Holding on to modularity, it's not a good idea to extend the parser program itself, so this functionality is implemented with a shell script that runs the parser on all the test cases in a directory. The script recursively traverses the given directory and counts successes and failures of every parsed file. Based on that information, it finally prints out absolute and relative success numbers for the files in the directory.

After removing invalid files, the following statistics have been retrieved for the test cases mentioned above:

Project	Total number of files	Successful passes	Failed passes
Eclipse	33038	33012 (99.9%)	26 (0.1%)
Hibernate	1474	1474 (100%)	0 (0%)
gcj libjava	4630	4630 (100%)	0 (0%)
ANTLR	314	314 (100%)	0 (100%)
zdt	143	143 (100%)	0 (0%)
mAkStats	40	38 (95%)	2 (5%)
HICS	34	31 (91.1%)	3 (8.9%)
Total	39673	39642 (99.92%)	31 (0.08%)

Table 2: Success ratio of test case parsing

The errors in mAkStats and HICS are due to the ISO-8859-1 encoding of the files and occur because of the special characters (umlauts and spanish characters with accents) that are not encoded in Unicode. When converting these files to Unicode, the errors disappear.

The errors in Eclipse fall into one of the two following categories:

Unicode escapes that are not preprocessed before lexing the input. This issue is covered in 4.3.2 and is done on purpose as a compromise between speed, effort and outcome. Furthermore, all the concerned files are test cases for Eclipse, which stengthens the theory that this is not needed by real-world programs.

The file size is still too big for the 500 Kilobyte file buffer that the parser's example framework provides. Upon integration with another framework like KDevelop's, this issue will disappear as well.

It is notable that 10 files in the Eclipse sources need the Java 1.4 compatibility mode and another 11 ones even require Java 1.3 as maximum version. When configured correctly (as done for the given statistics) they pass the parser runs successfully.

5 Future improvements

Although the parser's recognition quality is convincing, there is still a lot of work to do that can bring further benefits to maintainability, reusability and other qualities.

5.1 Improvements in kdevelop-pg

5.1.1 Support for class and struct members

Most of the status variables that are used in the Java parser have been defined as global static variables, like the one determining the Java version (described in section 4.5.3) or the `ltCounter` variable storing the number of open type argument brackets (described in section 4.5.1). This works sufficiently well, but has several drawbacks: Memory is allocated even if the parser is not used, the usage of the `static` keyword makes it impossible to run two independent parser instances at the same time, but worst of all is that this approach subverts the principles of object-oriented programming.

A similar issue is the lack of custom defined members inside the AST node structures. In several places, the AST could be simplified if it was possible to replace auto-generated members by self-defined ones. A good example can be found in the `mandatory_declarator_brackets` rule from the Java grammar:

```
   !( #lbracket=LBRACKET RBRACKET )
-> mandatory_declarator_brackets ;;
```

Here, a sequence annotation is used to count the number of declarator brackets. It would be preferable to define a `bracket_count` member inside the `mandatory_declarator_brackets_ast` node structure and to modify this rule so that it looks like this:

```
   !( LBRACKET RBRACKET [: (*yynode)->bracket_count++; :] )
-> mandatory_declarator_brackets ;;
```

Other places where custom defined members can be beneficial are, for example, modifier sequences (where the modifiers could be represented by a flag variable) or operators (where the member could store an enumeration value instead of a full-fledged token). Together these modifications can make AST structure processing easier.

kdevelop-pg could introduce additional code segments in the grammar file syntax to provide a solution for these situations. The idea is to be able to insert code into different places than what's possible now. In addition to the existing code insertion capabilities, the new ones could be used to insert code directly into the parser class declaration, into AST node structures or at the top of the .cpp file containing the implentation of the parser. The following code block is a syntax proposition for these additional capabilities, occuring at the beginning of a grammar file:

```
global/header [:
    class java;
    bool lookahead_is_cast_expression(java* parser);
:] ;;

global/implementation [:
    #include "java_lookahead.h"
    bool lookahead_is_cast_expression(java* parser) {
        java_lookahead* la = new java_lookahead(parser);
        bool result = la->is_cast_expression_start();
        delete la;
        return result;
    }
:] ;;

member/parserclass [:
public:
    enum java_compatibility_mode {
        java13_compatibility = 130,
        java14_compatibility = 140,
        java15_compatibility = 150,
    };

    java_compatibility_mode _M_compatibility_mode;
:] ;;

member/parserclass/constructor [:
    _M_compatibility_mode = java15_compatibility;
:] ;;

member/mandatory_declarator_brackets [:
    int bracket_count;
:] ;;
```

The implementation for this functionality requires a few modifications to kdevelop-pg's grammar file parser and some additional program logic, but should be very straightforward to write.

5.1.2 Support for unbounded LL(k) look-ahead

Section 4.5.2 describes a technique to implement unbounded LL(k) look-ahead for resolving certain ambiguities in the grammar. This approach is certainly feasible if the grammar is not too extensive or if only parts of the grammar are needed for performing the look-ahead. Nevertheless, a better approach would be automatic generation of the look-ahead code, combined with grammar file syntax extensions that allow to make use of this facility.

One way to accomplish this goal is an implementation of syntactic predicates. Generating a lookahead class like the one that was written for the Java parser requires only minimal modifications to the existing code generation routines. Other approaches include linear-approximate look-ahead [4] and full LL(k) look-ahead computation, the latter of which is less practical as it scales exponentially with the number of defined tokens [5].

5.1.3 Support for exiting loops

Section 4.6.1 discusses how to overcome missing loop exiting features using code segments. In the long run, we rather want this functionality incorporated into the parser generator itself to decouple grammar syntax from the knowledge of how the generated implementation looks in detail. A feasable solution would be an exit operator that is translated to the appropriate language construct inside the parser class. The syntax could be quite similar to the current one, even the `break` keyword seems to fit nicely. The main difference is that the author of the grammar no longer has to check the correctness of the generated code, as this task would then be assigned to kdevelop-pg.

```
        #name=identifier
        (  !( DOT ( #name=identifier | star=STAR {break} ) )
         | 0
        )
     -> qualified_identifier_with_optional_star ;;
```

A possible implementation has to take care to keep the FIRST set of the alternation item in a correct state, e.g. dismissing all tokens following the `{break}` item in the current subrule and viewing the FIRST set of this item as an epsilon rule.

5.1.4 Rule arguments

Currently, parsers generated by kdevelop-pg represent context-free grammars. In order to make it easier for rules to pass information to sub-rules or parent rules, kdevelop-pg could be extended with rule arguments. For example, let's take a look at `parameter_declaration_list` and its sub-rule `parameter_declaration_tripledot`:

```
        LPAREN [: tripleDotOccurred = false; :]
        (
           #parameter_declaration=parameter_declaration_tripledot
           @ ( 0 [: if( tripleDotOccurred == true ) { break; } :]
               COMMA
             )
         |
           0
        )
        RPAREN
     -> parameter_declaration_list ;;

        parameter_modifiers=optional_parameter_modifiers
        type_specification=type_specification
        ( triple_dot=TRIPLE_DOT [: tripleDotOccurred = true; :] | 0 )
        variable_identifier=identifier
        declarator_brackets=optional_declarator_brackets
     -> parameter_declaration_tripledot ;;
```

To pass the `tripleDotOccured` variable between both rules, it is currently necessary to declare it as a global variable or as an instance member of the parser class. With rule arguments it would be possible to make tripleDotOccured a local variable of the `parameter_declaration_list` rule instead, and to pass a pointer to the `parameter_declaration_tripledot` rule so it can be modified there. This would result in better readability and fewer unneeded public variables.

This would be easy to implement by verbatim copying of the specified arguments into the parameter list of the corresponding `parse_*()` method in the parser class and requiring every usage of such a rule to pass an argument.

An implementation that makes default values possible would be a bit more difficult as verbatim copying of parameter declarations with default values is only allowed in the class header but not in its method definitions. Adding this capability requires a syntactic seperation of parameter declarations and default values in the grammar, so that they can be processed seperately.

5.1.5 Definition of allowed tokens and the start rule

In its current state, kdevelop-pg generates an enumeration of integer values representing each token that occurs in the input grammar. While this seems to be intuitive at first glance, it proves to be a minor obstacle to the process of creating the grammar and the lexical analyzer.

In most cases the lexical analyzer is created first, so the author just wants to define a simple list instead of a full-fledged rule which is not even used in the grammar itself. If this rule is omitted and the grammar is still incomplete then the lexical analyzer is missing token definitions that are needed for the program to compile.

While these cases just create additional burden for the developer, the more dangerous effect of implicitly defined token lists is the fact that wrong token names inside the grammar are accepted as well. For example, if the developer makes the error of using “LT” as token name while “LESS_THAN” is used throughout the rest of the grammar, then both tokens are defined in the enumeration and are handled differently. Supposing the lexical analyzer only returns “LESS_THAN” tokens, every rule containing “LT” instead will fail.

In order to prevent such bugs, kdevelop-pg needs a construct for pre-defining all possible tokens that are allowed inside the grammar. If one of the rules contains a token which is missing in that list there will be an error indicating this issue. Implementing such a list is trivial, the greatest difficulty here might be deciding on a proper syntax.

What could also be useful is an explicit definition of the start rule. Currently a small part of parser logic must be implemented by the caller for gaining full parser functionality. This is what is needed in the Java parser's main.cpp (which is not part of the actual parser) to call the parser:

```
compilation_unit_ast *ast = 0;
bool matched = parser.parse_compilation_unit(&ast);
if (matched)
{
    // perform post-processing tasks, like calling visitor classes
}
else
{
    parser.yy_expected_symbol(java_ast_node::Kind_compilation_unit,
                              "compilation_unit");
}
```

Before this is done, the caller has to tokenize the input first by manually executing the lexical analyzer. In comparison, KDevelop's hand-written C++ parser contains all of the required logic inside the parser class which even encapsulates lexer calls. Calling the C++ parser from outside thus comes down to one single line:

```
TranslationUnitAST *ast = parser.parse( contents, size, m_memoryPool );
```

Parser usage could be made easier by requiring a start rule to be defined inside the grammar file. kdevelop-pg would then generate a `parse()` method for the parser class that initializes the needed variables and calls the `tokenize()` and `parse_start_rule()` methods (in case the start rule is called “`start_rule`”), also the call to the error recovery method `yy_expected_symbol()` will be integrated in the `parse()` method. `tokenize()`, like the error recovery methods, would be left to the caller to implement, it would only be declared inside the generated class declaration.

5.2 AST structure

One big advantage of KDevelop's hand-written C++ parser over the generated Java parser is the structure of its abstract syntax tree which is by far superior in terms of clearness and reusability. The C++ parser's AST does not resemble the path followed by the parser's recursively descending methods, instead it focuses on ease of use for further processing.

By generating the AST from grammar rules, kdevelop-pg can't provide these advantages. When using ANTLR, this is the point where tree parsers are used to post-process the original AST to create a new, clearer structure out of it. Something similar will be needed for the Java grammar too in order to make it accessible enough for a bigger number of developers. In any case, a class derived from the default visitor will be used to traverse the AST and extend the visitor's methods to store the information in another structure. Whether this structure will be an intermediate replacement AST or KDevelop's class model is yet to be determined.

In later versions, kdevelop-pg could be extended to support the creation of abstract syntax trees that differ from the structure of the grammar's rules.

5.3 Error recovery

While the Java parser operates smoothly on files that are syntactically correct, it lacks error recovery capabilities that make it possible to parse the whole source document even if it contains syntax errors. Especially when slated for integration with an integrated development environment, tolerance for input files is an important feature.

kdevelop-pg makes it possible to correct errors by implementing the `yy_expected_token()` and `yy_expected_symbol()` methods of the parser class. This way, simple error recovery can be done by skipping tokens until an expected token is found. KDevelop's hand-written C++ parser employs this technique, more precisely, it implements methods like `skipUntilStatement()` that dismiss all tokens but the ones defined therein. (This behaviour is also known as "panic mode" which is fast and easy to implement but, by design, ignores all possible errors up to the recovery point.) Observing these methods in detail shows that the "stop tokens" which cause the parser to return to its normal parsing state are comprised of the combined FIRST and FOLLOW sets of the rules in question.

One feasible approach for kdevelop-pg would therefore be automatic generation of such `skip*()` methods. The grammar file would add a syntax element to specify rules that should be recovered this way, the generated method for each given rule would recover on one of the tokens from the FIRST and FOLLOW sets and skip all other tokens.

More complex possibilities for automatic error recovery involve replacing errorneous tokens or inserting missing ones in order to correct the input token sequence. This area is well researched, possible approaches are for example the ones given in [6], [7] or [8].

5.4 Integration into KDevelop

The parser has been written to improve Java support in KDevelop. Having a complete lexical analyzer and grammar, the next step would be integrating the parser with the language support part.

KDevelop's language support is plugin-based – depending on the project type, the appropriate plugin is loaded. A minimal language support plugin consists of classes derived from the `KDevLanguageSupport` and `KDevGenericFactory` and a `kdev*support.desktop` file indicating the existance of the plugin to KDevelop (where "`*`" stands for the name of the respective language).

The parser is integrated by calling it from a `*ParseJob` class which is executed on every change of a source file in the project. After calling the generated parser, the `*ParseJob` class also runs the `*Binder` class which is responsible for transforming the AST into a `CodeModel` structure. The `CodeModel` holds all necessary information about scopes, classes, variables and the likes, and `CodeModel` structures are the way for KDevelop to interface with the contents of the source files. Once this structure is created, it is easy for KDevelop to provide advanced features like code completion or class overviews.

Also, the parser can be extended to report problems that were found in the code. Those are stored in a list of `Problem` structures and are later shown by the graphical user interface.

In KDevelop4, other language-dependent features like debuggers, build systems or class wizards are implemented in seperate plugins. This is done differently than in KDevelop3 where these features were coupled.

6 Conclusions

This document has shown the steps that are necessary to implement a Java parser based on the language specification and another already available grammar as reference. All of the required steps pose certain challenges to the parser developer, with practical solutions available for each of the encountered problems. Although it is feature complete in respect to parsing correct Java sources and generating the corresponding abstract syntax tree, there are still many possibilities left to improve the parser and integrate it with the KDevelop IDE so that it can actually be beneficial for real world development.

Speaking for myself, I have greatly enjoyed working on the parser. Not only did I learn much about theory and practical application of parser creation, I was even able to do so within an existing development community which produces one of the leading development environments for Linux/Unix systems. It is a great pleasure to know that my work will endure as part of a broadly used application, which has likely been the best possible way to stimulate my interest on language recognition topics and motivates me to continue this effort also after finishing this thesis.

7 Acknowledgements

I would like to thank Roberto Raggi, Adam Treat, Ian Geiser and Alexander Dymo for their gentle introduction to the KDevelop development community, Professor Franz Puntigam for providing the opportunity to realize this project, Martin Stubenschrott for his inspirations on the topic and help with LaTeX, and my parents, Christa and Gerhard Petsovits, for their ongoing moral and financial support thoughout my studies and my life.

References

[1] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

[2] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling. Volume I: Parsing.* Prentice Hall, 1972.

[3] James Gosling, Bill Joy, Guy Steele and Gilad Bracha. *The Java Language Specification, Third Edition.* Addison-Wesley Professional, 2005. (also available in digital form from http://java.sun.com/docs/books/jls/)

[4] Terence J. Parr. *Obtaining Practical Variants of LL(k) and LR(k) for k ¿ 1 by Splitting the Atomic k-Tuple.* PhD thesis, Purdue University, August 1993.

[5] D.J. Rosendrantz and R.E. Stearns. Properties of Deterministic Top-Down Grammars. In *STOC '69: Proceedings of the first annual ACM symposium on Theory of computing*, pages 165-180. ACM Press, 1969.

[6] P. van der Spek, N. Plat and C. Pronk. Syntax error repair for a Java-based parser generator. In *ACM SIGPLAN Notices*, Volume 40, Issue 4, pages 47-50. ACM Press, April 2005.

[7] Michael J. Burke and Gerald A. Fisher. A practical method for LR and LL syntactic error diagnosis and recovery. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Volume 9, Issue 2, pages 164-197. ACM Press, April 1987.

[8] Susan L. Graham and Steven P. Rhodes. Practical syntactic error recovery. In *Communications of the ACM*, Volume 18, Issue 11, pages 639-650. ACM Press, November 1975.

[9] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional, 1999.

[10] Erich Gamma, Richard Helm, Ralph Johnson and Vohn Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 1995.

[11] The Unicode Consortium. *The Unicode Standard, Version 4.0.* Addison-Wesley Professional, 2003.

[12] ISO/IEC 14977:1996. *Information technology - Syntactic metalanguage - Extended BNF.* (also available in digital form from http://standards.iso.org/ittf/) PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996(E).zip)

[13] Martin Stubenschrott. *A context sensitive code completion system for the C and C++ programming languages.* Bachelor thesis, Institute for Computer Languages, Vienna University of Technology, April 2005.

[14] Terence J. Parr and Russell W. Quong. ANTLR: A Predicated-LL(k) Parser Generator. In *Journal of Software Practice and Experience*, Vol. 25(7), pages 789-810. John Wiley & Sons Ltd., July 1995.

[15] Stephen C. Johnson. *Yacc: Yet another compiler-compiler. Technical Report 32.* Bell Laboratories, 1975.